Cairo Coding Guidelines

Massil Achab

Follow

Nethermind.eth

--

1

Listen

Share

by

[Massil Achab

](https://twitter.com/machilassab)and

[Kalzak

](https://twitter.com/kalzakdev).

Thanks to

[Mathieu Saugier

](https://twitter.com/eniwhere_),

[Soufiane Hajazi

](https://twitter.com/quixotic_eth),

[Krzysztof Szubiczuk

](https://twitter.com/kaliberpoziomka)and

[Edgar Barrantes

](https://twitter.com/EdgarBarrantes)for helpful comments and revisions.

Guidelines to write Cairo

Writing Cairo can be challenging, mainly for two reasons: smart contract developers are tempted to use patterns from Solidity code, and Cairo is a constantly evolving language with new syntax and API changes every few months. Still, ZK-rollup technology is here to stay, and an increasing number of developers onboard every week to learn StarkNet and Cairo. At Nethermind, we have gained experience with Cairo across several projects. We want to share our list of patterns, tips, and recommendations that we use to standardize our Cairo code.

Foreword

As a team, following a fixed set of rules is a convenient way to maintain a consistent code style across a codebase. This requires all team members to commit to the same code style while writing and reviewing. A benefit to a consistent code style is improved readability, which is crucial during an audit process. The more readable your code is, the easier it is for auditors to understand your protocol, allowing them to focus on more technical security aspects.

Code

Composition over Inheritance

One of the many differences between Solidity and Cairo is that you can't use inheritance with Cairo. Instead, you can use the composition over inheritance principle — called "Extensibility pattern" by OpenZeppelin — to define a class by reusing the code of other classes. We propose a file structure at the end of this guide. More precisely:

for each contract, we define:

- A library file (or logic file), named my_contract_library.cairo

, contains the logic code of the contract. Namely, it contains: (i) internal and external functions encapsulated in a namespace, and (ii) storage variables and events defined outside the namespace.

- A contract file, named my_contract.cairo

, exposes external functions from its corresponding library file and other library files. For instance, an implementation of cToken that inherits from an ERC20 contract would expose both functions in c_token_library.cairo

and erc20_library.cairo

.

for each abstract contract, we define:

- A library file, following the same naming convention than above. Indeed, an abstract contract should not expose functions but provide code to other contracts.

Naming conventions

We follow most, but not all, recommendations from OpenZeppelin, detailed on their page aboutExtensibility pattern. More precisely:

- We use snake_case

for filenames, functions (in library files) and variables.

- We use snake_case

for view and external functions (functions in contract files).

- We use UPPER_SNAKE_CASE

for constants.

- We use PascalCase

for struct, namespaces, interfaces and events:

- We use PascalCase_snake_case

for storage variables, where the PascalCase

part is namespace's name:

Import order

Regarding Cairo imports, we group them into three categories separated by a newline and alphabetically sort each category. Sorting imports enables fast retrieval for the reader. The three categories are:

1. Starkware imports
2. External libraries imports (usually cairo-contracts from OpenZeppelin).
3. Internal code imports

Error messages

When using with_attr error_message(...)

statement, make sure only one expression in the statement can fail.

Otherwise, if an expression fails and the error message doesn't correspond, the developer can spend a lot of time trying to debug.

Calldata

With Cairo, to pass an array of felt to a function, the usual pattern is to pass a pointer of felt and the array's length. We recommend encapsulating this array in a struct MyStruct

and use MyStruct.SIZE

for the array length.

Otherwise, if you hardcode the array's length in a function — say IProxy.initialize(pool, 2, new (name, value))

- and want to add an element decimals

to the array, you may forget to increment the array length at the same time. Using structs for this is a way to use a dynamic length instead of a fixed number.

Guards

We recommend calling guards in the contract file. In Solidity, we usually define them with modifiers. Indeed, calling it in the library file can result in reusing it without doing so on purpose.

Example:

Booleans

Since Cairo doesn't have a separate type for booleans, we use felts. It can be misleading for a newcomer to read the following statement: assert bool_1 + bool_2 + bool_3 = 3

. Instead, we recommend using a library for comparing and combining boolean variables, likeBoolCmp, for better readability.

Loops

Cairo doesn't provide support for loops. The alternative is to define recursive functions that operate on arrays instead. For each loop, we recommend defining an internal function suffixed with _inner

or _loop

to do the job.

Example:

Storage variables

Storage variable names could conflict for Cairo version prior to 0.10.0. In case of a conflict, it now raises an error at compilation time. We recommend prefixing storage variable names with the namespace to avoid a collision.

Asserts

One can assert several equalities at the same time using structs. Let's say that a function simple_func

returns three values a

, b

and c

. You can use the following:

Math

Cairo standard library currently provides misleading utils functions to compare felts; seethis issue, for instance. We have written two libraries to work safely with signed and unsigned felts: SafeCmp to handle felt comparisons and FeltMath for math operations.

Unsigned felts

Cairo's native integer type belongs to a finite field of order P = 2^251 + 17 * 2^192 + 1

. A way to think about it is to apply integer division by P

after every math operation. In particular, integers can take values from 0

to P-1

and then go back to 0

. That is: P = 0

is a true statement in Cairo.

Thus, one can represent unsigned felts with values {0, 1, …, P-2, P-1}

. If a math operation’s result is strictly greater than P-1

(resp. strictly lower than 0

), it can be considered an overflow (resp. an underflow).

Signed felts

Cairo natively supports signed felts by casting them to {0, 1, …, P-2, P-1}

. Recall that P = 0

in Cairo, then -1

is simply regarded as P-1

. With that in mind, we can uniquely represent integers of the interval {-(P-1)/2, -(P-1)/2 + 1, …, (P-1)/2–1, (P-1)/2}

using felts. In particular, it means that the maximal signed felt is (P-1)/2

, and that the minimal signed felt -(P-1)/2

is represented by P-(P-1)/2 = (P-1)/2 +1

.

The two libraries mentioned above are built on top of these findings.

Contract structure example

Library file

my_contract_library.cairo

Contract file

my_contract.cairo